V.08.

Общий обзор занятия.

Переходное занятие получается, хотя переход получается вполне себе естественный, гладкий такой. И это хорошо. Переходим к рекурсии. И в ближайшие несколько занятий будем говорить именно о рекурсии, о хорошем и плохом, о том, как с ней обращаться и всё такое просее... Один из главных моментов, конечно, и это, скорее всего, самый главный в блоке про рекурсию - технический момент: освоиться с ней, научиться не трассировать рекурсию, а смотреть на неё естественно - “сверху вниз”, не влезая в частности трассировки, даже не пытаясь это сделать. Главное - понять, где же у рекурсии эти частности, мелочи, влияющие на весь механизм её действия. Совсем по-простому говоря, научиться с ней обращаться - тут несколько другой взгляд нужен, не технология, а именно взгляд, но несколько иной взгляд, с иной стороны. И научиться видеть, чувствовать рекурсию - это основной момент на весь блок рекурсии. О свойствах, об особенностях - да, это образует всё содержание, но это - внешнее, это знания, конкретные знания, очень важные, как любые конкретные знания, но это всё должно подводить именно к пониманию рекурсии, к тому, из чего они все, эти свойства, вытекают, к понимаю сути. Понимание рекурсии должно быть внутри всего блока, именно оно определяет все акценты, течения, направления. Именно это есть движитель всего процесса, сверхзадача его.

Ну, и после такого несколько пафосного вступления, немного больше конкретики. У нас накануне было довольно много про сортировки, хоть и туповатые. Вот и продолжим эту линию - строим бинарное дерево поиска. Структура данных, чем-то схожа со списками, но принципиально иная - она не линейная, данные не вытягиваются в линеечку (честно говоря вытягиваются, но это совсем не очевидно, и не выглядит совсем уж естественным, но это так, издержки, ибо “Во многой мудрости много печали; и кто умножает познания, умножает скорбь”. (Екк. 1:18)). Строим постепенно, и довольно быстро наталкиваемся на рекурсию, подчеркну, наталкиваемся совершенно естественно, деревья и рекурсия - близнецы-братья. И на этом занятии мы очень естественным образом, влёгкую, разбираемся с рекурсией, причём даже и в некотором разнообразии. Правда полностью разложить всё с деревом поиска не получится -

и сложновато для первого раза, и не нужно пока, но с удалением элемента и с балансировкой дерева мы разбираться будем, но существенно позже (но в этом семестре). И после сегодняшнего занятия на несколько занятий уходим в разговоры о рекурсии во всяких ситуациях и вариантах. А вот когда мы с ней более или менее обустроимся, когда дети пооботорутся хоть немного, когда они себя в рекурсии почувствуют, тогда мы и вернёмся снова к структурам данных, и поговорим тогда о паре-тройке схем балансированных деревьев, о более сложных операциях с деревьями, возможно ещё о чём-нибудь, поживём-увидим...

Лекция

План такой:

- заводим дерево поиска совсем по-простому:
  - каждый узел хранит два указателя,
  - при вставке нового числа (элемента, вообще говоря, но мы будем играться в примерах с числами, так что я для простоты буду говорить про числа, хотя это совешенно не существенно) числа, меньшие того, что находится в текущем узле идут налево, бОльшие - направо
- вставляем новый элемент, получается очень легко, безо всякой рекурсии
- не менее легко и не более с рекурсией пишем поиск элемента - ответ на запрос, есть данное число в дереве или нет
- но раз уж мы числа как-то сравнивали, то начинает ощущаться легкий мотив сортировки, и мы говорим, что хорошо бы это использовать, авось сортировка и получится; пытаемся вывести числа дерева в порядке возрастания.
- и наталкиваемся на ситуацию “близок локоть, да не укусишь” - вроде всё так понятно: да, сортировка видна, но сформулировать правила обхода полученной структуры - бинарного дерева поиска - никак не получается.
- и тут на свет выступает рекурсия
- при этом мы связываем рекурсивный обход с рекурсивным определением дерева
- и всё срастается!
- а дальше мы радостно переделываем вставку и поиск на рекурсивный манер

- ну, и ещё каких-нибудь операций забрасывем, типа вычисления глубины дерева или подсчёта количества листьев

В итоге мы довольно естественно переходим от нерекурсивных методов обрабоки к рекурсивным.

В качестве бонуса рассмотрим такую задачу: найти K-е (K - заданное число) по порядку число - вроде сортировка есть, значит можно и K-е число можно искать как-то получше. Но оказывается, что для этого приходится перебирать все предыдущие числа, а это тормозит. И тогда мы изменяем немного структуру - в каждый узел добавляем размер поддерева, корнем которого является это узел (рекурсивное определение дерева!). И получается просто сказка.

Вопросы удаления элемента, возможного дисбаланса и балансировки дерева касаемся еле-еле, причём в контексте “это не так просто, об этом попозже, но это возможно, но об этом потом”.

Вот такой план.

Основная идея в этот раз – не просто показать рекурсию в первый раз, а показать рекурсию с «лёгкой», красивой стороны, не показывать проблемы рекурсии, обходить опасные и сложные моменты. Главное – чтобы детям рекурсия понравилась. Достигается это за счёт того, что мы всячески пресекаем попытки трассировать рекурсивные вызовы, хотя дети будут пытаться именно это делать. Уводить их от таких попыток всеми возможными способами и средствами. Всё рассматриваем только «сверху». Типа, ну, вот надо что-то сделать, как-то обработать объект. Обрабатываем одну часть, другую, может быть, третью, четвёртую,... и получаем что мы исполнили нужную обработку со всем объектом. Или делаем некоторуое действие и приходим к тому, что надо выполнить ту же самую обработку но уже с «меньшим» объектом, на другом, более лёгком уровне. Ну, вот хотя бы: чтобы подняться по лестнице надо подняться на одну ступеньку, а потом подняться по лестнице, но уже меньшей. Вообще, вот такой индуктивный подход очень тут уместен.

Рекурсия – это легко. А вопрос, почему же это работает, рассматриваем только “сверху”: вот, обработали вот это, потом вот это, и, раз наша процедура должна, судя по всему, делать то, что нам надо с бОльшим объектом, то она правильно работает для меньших объектов. А как она работает для меньших объектов? Правильно? Значит, мы же её так писали, она будет правильно работать для

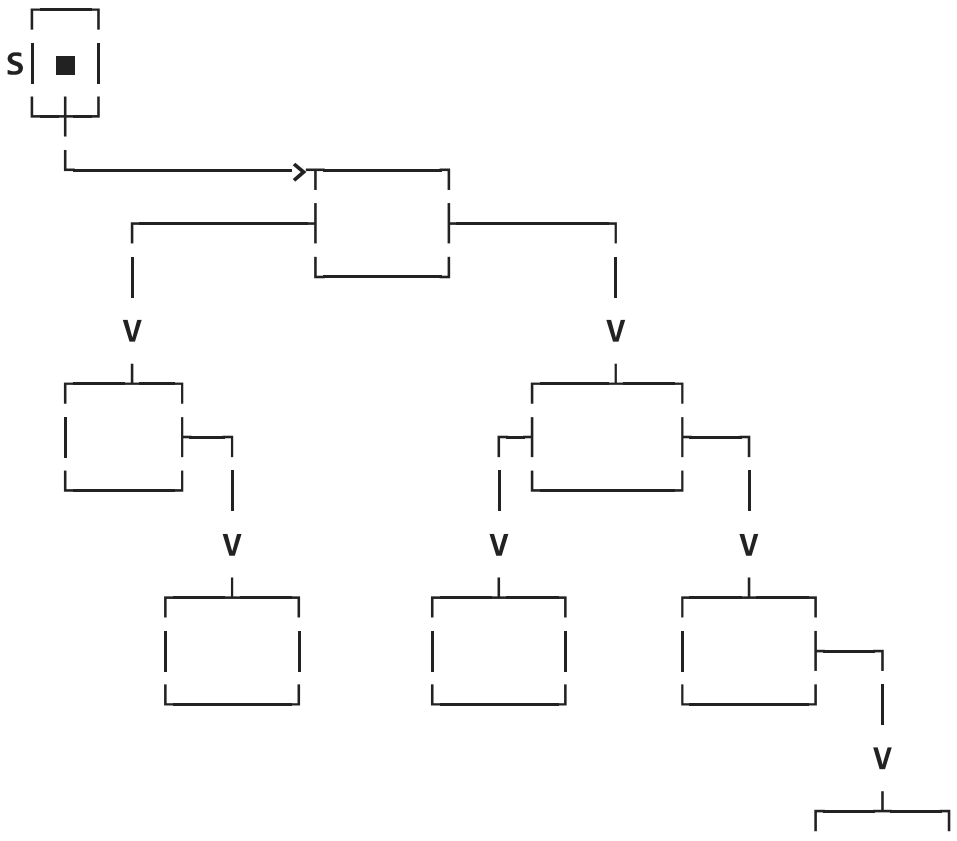
бОльших объектов. А порочный круг? А нет порочного круга – потому, что у нас корректно обрабатывается концовка - самый маленький, неделимый объект.

Много слов, но такой момент - вступление к большой сущности.

Поехали.

Начинаем строим бинарное дерево поиска.

И начинаем мы его строить очень так по простому. Ну, то есть объясняем детям, что вот у нас будет теперь двоичное дерево. Объясняем, что такое двоичное дерево. Т.е. простенькое такое определение, не совсем формальное: типа, двоичное, оно же бинарное, дерево – это такая структура, что у каждого узла (кроме одного – корня – за который мы всю конструкцию и держим) имеется ровно один предок, из которого в этот узел идёт стрелочка, и от 0 до 2 детей, в которые из узла идут стрелочки. Такая терминология - дети-родители, отцы-сыновья - идёт от генеалогических деревьев, а не от тех, которые растут в земле, поэтому, видимо, и принято рисовать деревья корнем вверх, листьями (а это вершины без детей) вниз. Получается примерно вот такая вот структура:



Соответственно, конструируем типы для хранения бинарного дерева и узла дерева:

```
type  (
    BinaryTree struct
        root *BinaryNode
    }
    BinaryNode struct {
        key keyType
        lson *BinaryNode
        rson *BinaryNode
    }
)
```

`key` - это ключ, ключ сортировки. Дело в том, что мы строим дерево поиска. А главное свойство бинарного дерева поиска в том, что все потомки слева от любой вершины содержат ключи, меньшие, чем ключ в самой вершине, а все потомки справа содержат значения не меньшие, чем ключ в этой вершине. Понятно, что данные в каждом узле должны обладать ключами, на которых определена операция сравнения меньше. Зачастую информация, представляющая данные в каждом узле, является стркутурой, а не единственным полем данных. Однако это касается реализации, а не природы двоичного дерева поиска. Мы будем считать для начала, что `keyType = int` .

Я повторю всё это ещё раз, чуть более формально - приведу цитату из Вики - `https://ru.wikipedia.org/wiki/Двоичное_дерево_поиска` .
цитата:
Для целей реализации двоичное дерево поиска можно определить так:

- Двоичное дерево состоит из узлов (вершин) — записей вида (data, left, right), где data — некоторые данные, привязанные к узлу, left и right — ссылки на узлы, являющиеся детьми данного узла — левый и правый сыновья соответственно. Для оптимизации алгоритмов конкретные реализации предполагают также определения поля parent в каждом узле (кроме корневого) — ссылки на родительский элемент.

- Данные (data) обладают ключом (key), на котором определена операция сравнения «меньше». В конкретных реализациях это может быть пара (key, value) — (ключ и значение), или ссылка на такую пару, или простое определение операции сравнения на необходимой структуре данных или ссылке на неё.
- Для любого узла X выполняются свойства дерева поиска: key[left[X]] < key[X] ≤ key[right[X]], то есть ключи данных родительского узла больше ключей данных левого сына и нестрого меньше ключей данных правого.

конец цитаты

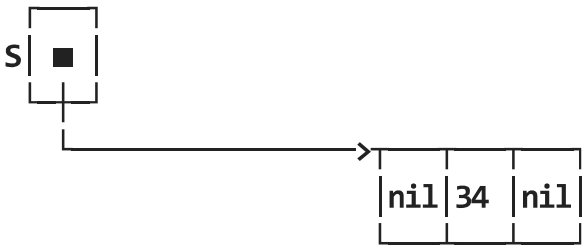
А как обеспечить выполнение свойств дерева поиска?

Ответ простой - вставлять новые элементы последовательно, сразу в процессе вставки поддерживая эти свойства.

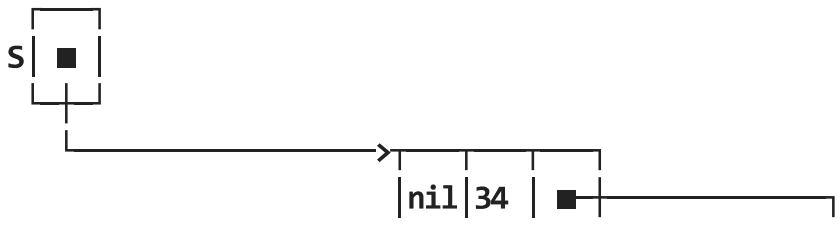
Давайте посмотрим, как это работает.

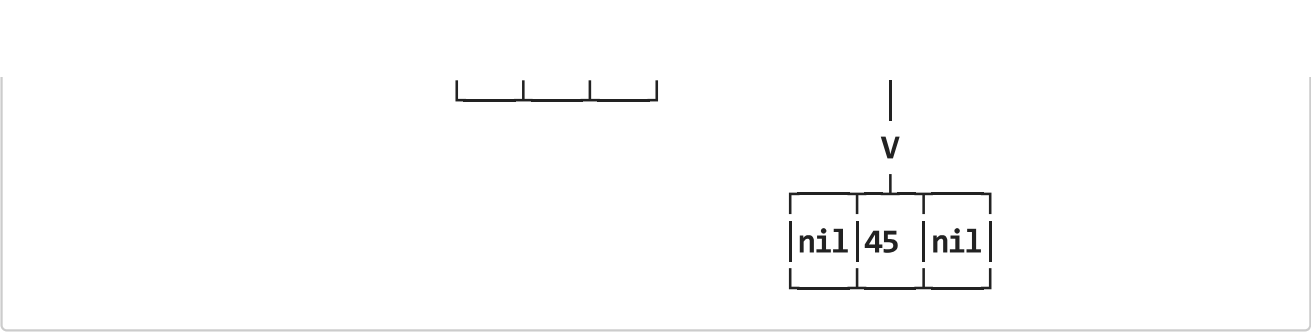
1. Начало. Пустое дерево поиска.

2. Вставляем число 34.

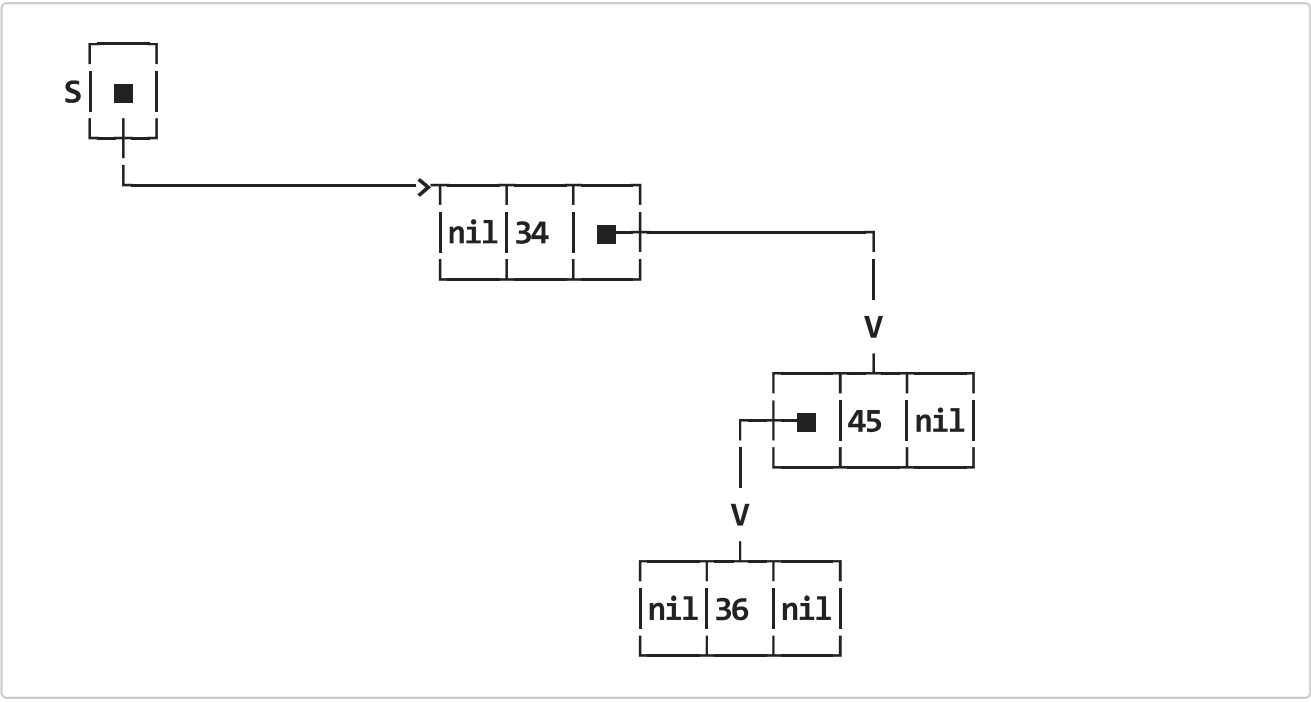


3. Вставляем число 45.

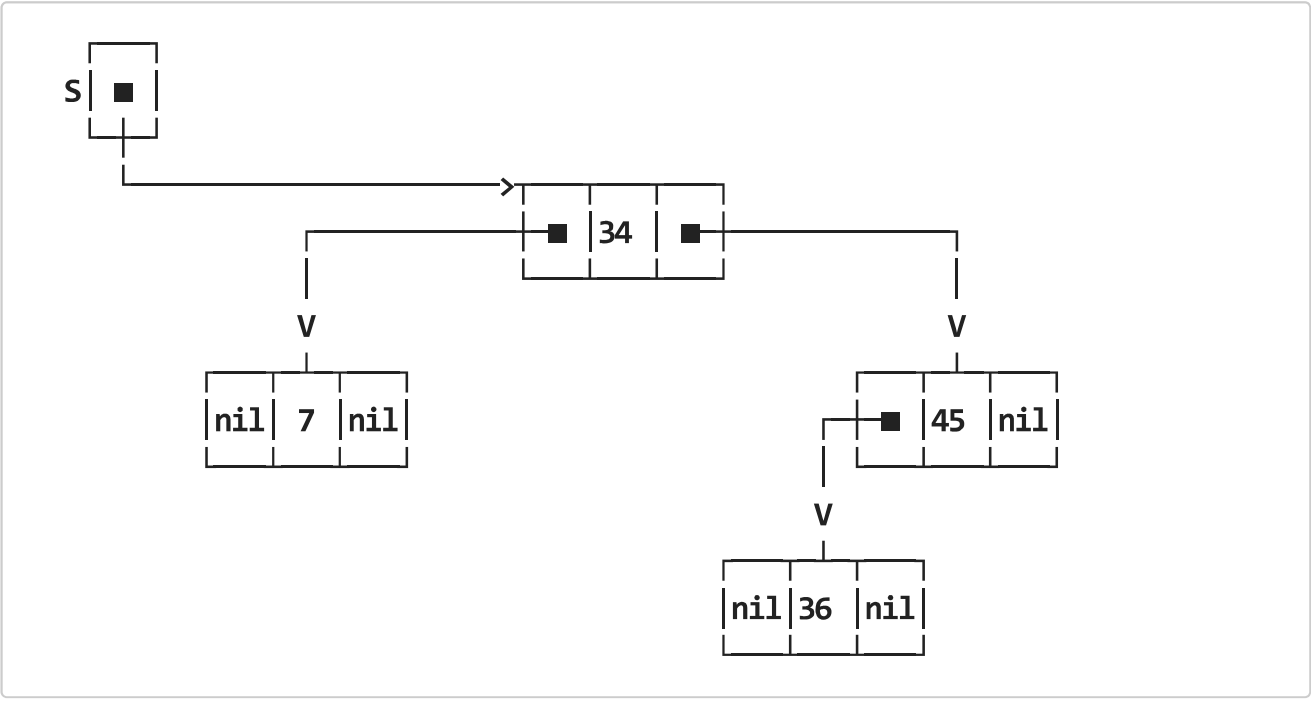




4. Вставляем число 36.

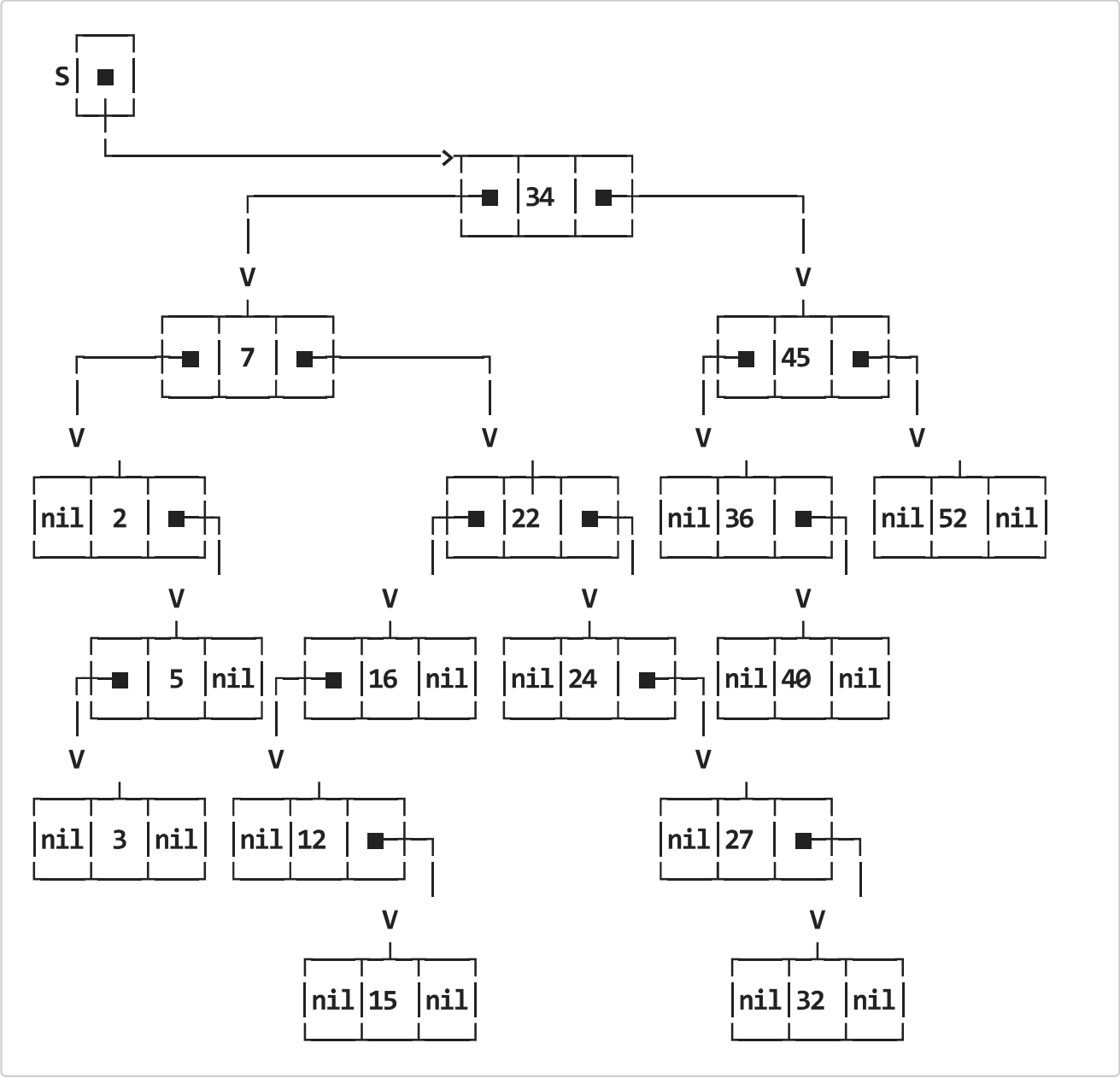


5. Вставляем число 7.



Ну, и так далее. Процесс более или менее понятен. Вставим

последовательно числа (ключи) 22, 24, 2, 40, 27, 5, 3, 52, 16, 32, 12, 15 и получим вот такой вот шедевр:



На всякий случай проговорю правила добавления новых ключей. Итак, у каждого узла от 0 до 2-х детей, причём слева от каждого узла все потомки (не только дети) содержат меньшее значение ключа, а все потомки справа – не меньшее. И новые ключи пристраиваем соответствующим образом: идём по дереву, новый ключ (число) меньше ключа в узле – идём налево, иначе – идём направо. Когда некуда идти (ну, нету узла в соответствующем направлении), то пристраиваем в нужном направлении новый узел, и запихиваем в него новый ключ, а указатели налево и направо в нём делаем равными nil, чтобы можно было как-то потом понимать, что дальше хода нет.

Код вставки нового ключа. Пока ещё нерекурсивный.

пример 01.go.

```
package main

import  (
    "fmt"
)

type  (
    BinaryNode struct {
        key int
        lson *BinaryNode
        rson *BinaryNode
    }
    BinaryTree struct {
        root *BinaryNode
    }
)

func InitTree() BinaryTree {
    return BinaryTree{root: nil}
}

func (s BinaryTree) Empty() bool {
    return s.root == nil
}

func (s *BinaryTree) Insert(n int) {
    if (*s).Empty() {
        (*s).root = &BinaryNode {key: n, lson: nil, rson: nil}
        return
    }
    current:= (*s).root
    for  {
        if n < (*current).key  {
            if (*current).lson == nil  {
```

```
                (*current).lson = &BinaryNode{key: n, lson: nil, rson: nil}
                return
            }  else  {
                current = (*current).lson
            }
        }  else  {
        // n >= (*current).key  {
            if (*current).rson == nil  {
                (*current).rson = &BinaryNode{key: n, lson: nil, rson: nil}
                return
            }  else  {
                current = (*current).rson
            }
        }
    }
}

func (s BinaryTree) Search(n int) bool {
    current:= s.root
    for  {
        switch {
        case current == nil:
            return false
        case n == (*current).key:
            return true
        case n < (*current).key:
            current = (*current).lson
        case n > (*current).key:
            current = (*current).rson
        }
    }
}

func main() {
    data:= []int{34, 45, 36, 7, 24, 2, 40, 27, 5, 3, 52, 16, 32, 12, 15}
    tree:= InitTree()
```

```
    for _, key := range data {
        tree.Insert(key)
    }
    fmt.Println(tree.Search(12))   // true
    fmt.Println(tree.Search(18))   // false
    fmt.Println(tree.Search(3))    // true
}
```

Заодно сюда вставлен код поиска ключа - `func (s btree) Search(n int) bool` проверяет, есть ли в дереве ключ `n`.

Код ясный, совершенно прозрачный, кажется, в комментариях не нуждается. Да и особо зависать на этом месте не стОит - нам ещё предстоят подвиги в этот раз, так что берегите силы.

Обход дерева поиска.

И вот теперь, когда народ расслаблен и не ждёт подвоха, вот тут-то мы их – бац! – и просим вывести содержимое дерева. Ну, так, словами описать процедуру вывода ключей, содержащихся в дереве. Ой, что будет... Долго зависать на этом не надо, но дать детям немного помучиться вполне себе есть смысл, чтобы почувствовали задачу чуть поглубже. Если ничего связного в результате не родится (скорее весго, так и будет), дадим детям посказку: а давайте не просто выводить, а выводить числа (ключи) в порядке возрастания.

Что изменяется? Изменяется то, что нам придётся (и можно) использовать особенность именно дерева поиска, что в нём меньшие идут налево, остальные - направо, что при построении дерева поиска исполняются какие-то сравнения, т.е. как-то это построение связано с сортировкой. Т.е. предложение выводить в порядке возрастания - это действительно подсказка, а вовсе не издевательское усложнение задачи. Разумеется, первым делом дети предложат идти всё время влево до упора. Это ладно. А вот потом... Потом я не берусь предсказать, но вариантов будет немало. И вот, после того, как дети несколько покувыркаются, подчеркну, покувыркаются не сильно долго – чтобы успели проникнуться сложностью проблемы, но не успели заморочиться ею же, - мы их выводим на рекурсию. Здесь надо действовать очень точно.

Давайте теперь подумаем... Вот мы держим дерево за корень. Хочется его - этот корень - вывести. Но! Перед этим надо вывести что-то. Что? Все ключи, меньшие того, который в корне. А где они все? А вот - слева от корня. А что там у нас слева от корня? АААААААААААААААААЕЕЕЕЕЕЕЕЕЕ! Это дерево! Точнее, поддерево, но всё равно дерево. И надо вывести все его элементы в порядке возрастания. Как? Так вот взять и вывести, как только что выводили. А вот потом, после всех них, надо вывести число в текущем корне, а потом все числа, бОльшие этого числа, а они образуют правое поддерево корня. Ура? А порочный круг не получится, типа чтобы вывести дерево, надо уметь выводить дерево? А вот об этом не надо пока. На темпе проходим это склизкое место - мы же пока ещё код не пишем, мы пока только думаем.

Придумали алгоритм: чтобы вывести все ключи дерева в порядке возрастания надо:

1. Вывести все ключи в левом поддереве (дереве!) в порядке возрастания
2. Вывести ключ в корне
3. Вывести все ключи в правом поддереве (тоже дереве!) в порядке возрастания

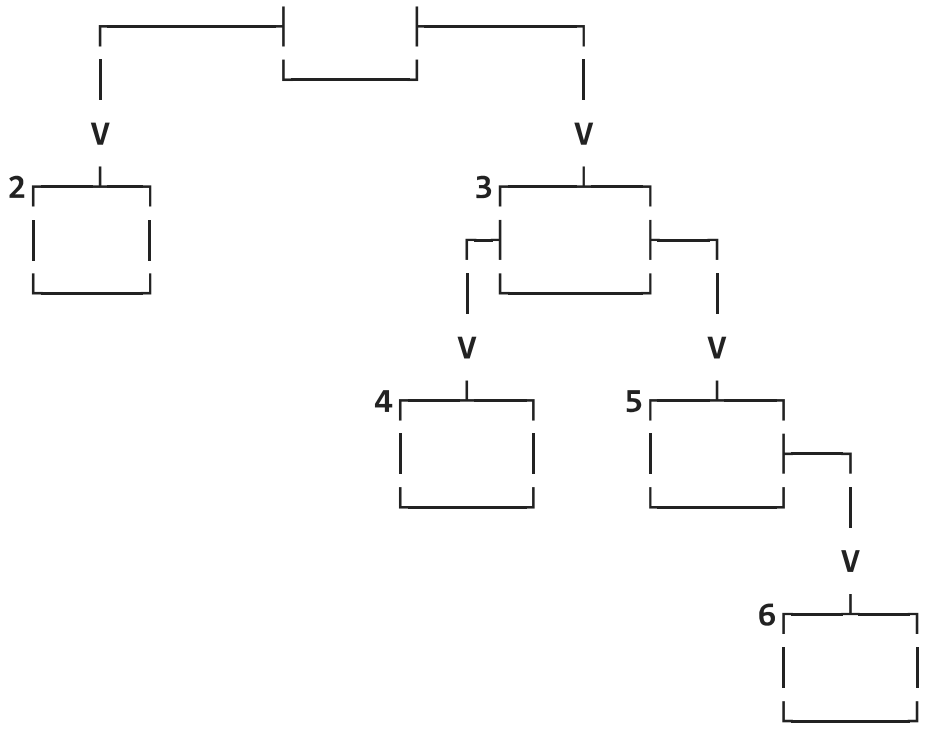
Бинарное дерево - рекурсивное определение.

И сразу в темпе, чтобы у нас был готов ответ на вопрос о порочном круге, даём вот такое
определение бинарного дерева:

Бинарное дерево - это

```
- пустое множество  ИЛИ
- одиночная вершина  ИЛИ
- вершина, к которой прилеплены слева и справа два непересекающихся бинарных дерева
```

И дальше поиграем в проверку определения, разбираемся, почему же нет порочного круга. Вопрос: вот эта картинка - это бинарное дерево? Цифрки на картинке - это просто нумерация вершин, чтобы было понятнее дальнейшее.



Конечно, да, скажут дети. А почему. А потому, что есть вершина (1) и слева к ней прицеплено бинарное дерево (2) и справа к ней прицеплено бинарное дерево (3-4-5-6). Почему то, что слева, есть бинарное дерево понятно - по определению: одиночная вершина есть бинарное дерево. А вот почему вершины 3-4-5-6 образуют бинарное дерево, пока непонятно. Ну, хотя бы понятно, что левая подвеска и правая не пересекаются.

Так что там с 3-4-5-6? Конечно, это бинарное дерево, скажут дети. И тут, по идее, они уже должны сами сказать, что слева висит вершина 4, а она есть дерево, потому что единственная вершина есть по определению бинарное дерево, а пара 5-6, во-первых, не пересекается с левой частью (4), а, во-вторых, есть бинарное дерево.

А почему, собственно? А потому, что к корню 5 подвешены два бинарных дерева: слева - пустое множество, а справа - одиночная вершина 6.

Да. Мы доказали. От этого места сильно повanивает трассировкой рекурсии, так что здесь увлекаться и пережимать не надо, тут всё понятно интуитивно, и надо бы эту ясность по возможности сохранить, хотя и формальным выкладкам тут есть место.

И порочного круга нет. Мы его разорвали. Как? А концовкой - мы не ушли в “у попа была собака...” благодаря тому, что по определению пустое множество и

одиночная вершина являются бинарными деревьями.

Наверно можно тут разобрать пару примеров на небинарные деревья или на недеревья (конструкции, у которых пересекаются левая и правая подвески), доводя именно до той точки, где, да, полученная конструкция не попадает ни под один из трёх случаев: левая и правая подвески пересекаются, либо из какой-то вершины вниз выходит 3 или 4 стрелки.

По ходу пьесы замечаем, что в приведённом определении упоминать одиночную вершину не обязательно, поскольку и без этого ясно, что одиночная вершина есть вершина, к которой слева и справа привешены два бинарных дерева - пустых множества.

Так что вот это вот **определение**
Бинарное дерево - это

```
- пустое множество  ИЛИ
- вершина, к которой прилеплены слева и справа два непересекающихся бинарных дерева
```

совершенно эквивалентно предыдущему.

Бинарное дерево поиска - рекурсия в реализации.

Спросим у детей, а как вывести ключи в порядке убывания? Должны сказать. Смотрим код (в нём, правда, Serach и Insert тоже езменены на рекурсивный вариант, но об этом сразу после просмотра кода обхода дерева - функции Trace): **пример** 02.go.

```
package main

import  (
    "fmt"
)

type  (
    BinaryNode struct {
```

```
        key int
        lson *BinaryNode
        rson *BinaryNode
    }
    BinaryTree struct {
        root *BinaryNode
    }
)

func InitTree() BinaryTree {
    return BinaryTree{root: nil}
}

func (s BinaryTree) Empty() bool {
    return s.root == nil
}

func (t *BinaryTree) Insert(data int) {
    if (*t).Empty() {
        (*t).root = &BinaryNode{key: data, lson: nil, rson: nil}
    } else {
        (*t).root.Insert(data)
    }
}

func (node *BinaryNode) Insert(data int) {
    if data < (*node).key {
        if (*node).lson == nil {
            (*node).lson = &BinaryNode{key: data, lson: nil, rson: nil}
        } else {
            (*node).lson.Insert(data)
        }
    } else {
    //  data >= (*node).key
        if (*node).rson == nil {
            (*node).rson = &BinaryNode{key: data, lson: nil, rson: nil}
```

```
        } else {
            (*node).rson.Insert(data)
        }
    }
}

/*   Вариант вставки ключа:
func (t *BinaryTree) Insert(data int) {
    insertnode (&((*t).root), data)
}

func insertnode(node **BinaryNode, data int) {
    if *node == nil {
        *node= &BinaryNode{key: data, lson: nil, rson: nil}
    } else {
        if data < (*node).key {
            insertnode (&((*node).lson), data)
        } else {
        //  data >= (*node).key
            insertnode (&((*node).rson), data)
        }
    }
}
Конец варианта */

func (s BinaryTree) Search(n int) bool {
    if s.Empty() { return false }
    switch {
    case n == (*s.root).key:
        return true
    case n < (*s.root).key:
        return BinaryTree{root: s.root.lson}.Search(n)
    case n > (*s.root).key:
        return BinaryTree{root: s.root.rson}.Search(n)
    }
    return true // или false - всё равно до этой строки не доходит
```

```
}


func (node *BinaryNode) traceUp()  {
    if node != nil  {
        (*node).lson.traceUp()
        fmt.Print((*node).key, " ")
        (*node).rson.traceUp()
    }
}

func (node *BinaryNode) traceDown()  {
    if node != nil  {
        (*node).rson.traceDown()
        fmt.Print((*node).key, " ")
        (*node).lson.traceDown()
    }
}

func (s BinaryTree) Trace(dir int)  {
    // dir > 0 - выводит в порядке возрастания
    // dir < 0 - выводит в порядке убывания
    switch  {
    case dir > 0:
        s.root.traceUp()
    case dir < 0:
        s.root.traceDown()
    }
    fmt.Println()
}


func main() {
    data:= []int{34, 45, 36, 7, 24, 2, 40, 27, 5, 3, 52, 16, 32, 12, 15}
    tree:= InitTree()
    for _, key := range data {
```

```
        tree.Insert(key)
    }
    fmt.Println(tree.Search(12))   // true
    fmt.Println(tree.Search(18))   // false
    fmt.Println(tree.Search(3))    // true
    tree.Trace(1)
    tree.Trace(-1)
}
```

Абсолютно с тех же позиций рассматриваем теперь (рекурсивные!) функции и Search и Insert. Search идейно вообще ничем не отличается от Trace: в пустом дереве ключа нет, там вообще ничего нет; если же ищем в непустом дереве, то смотрим на корень - если корень равен ключу, который ищем, то сразу возвращем true, иначе ищем там, где ещё есть надежда найти это ключ – то ли в левом поддереве, то ли в правом. А вот в Insert вроде всё то же самое: если вставляем в пустое дерево, то ставим в его корень новый элемент; если же вставляем в непустое дерево, то в корень поставить нельзя – смотрим, куда вставить новый элемент – в левое поддерево или в правое поддерево – и вставляем его в соответствующее поддерево. Но при этом есть некоторые отличия. Дело в том, что тип дерева не есть просто указатель на узел, а есть структура, содержащая единственное поле - указатель на узел, на корень дерева в данном случае. Это приходится делать для того, чтобы можно было писать методы дерева. И поэтому мы пишем два метода Search: один для BinaryTree, другой - для BinaryNode. Почему в Search и Trace обошлись без этого? А потому, что Insert изменяет свой параметр-ресивер, от него передаётся адрес. Ну, проще тут посмотреть на код. И мы это делаем, спокойно и тщательно.

А там, кроме того, есть и ещё один вариант вставки нового ключа. И, мне кажется, что он даже более прозрачен, хотя и закомментирован. Но в нём есть несколько тяжеловатый для детей момент - мы передаём там указатель на узел (величину типа *BinaryNode)* по адресу. Поэтому и появляются * в заголовке функции

```
func insertnode(node **BinaryNode, data int)
```

Да, и название написано с маленькой буквы, хотя это здесь совсем неважно. Но так сделано для того, чтобы подчеркнуть чисто реализационный характер этой функции. Т.е. да, мы берём и изменяем дерево (в том числе и поддеревья при

рекурсивных вызовах), т.е. поле root в структуре типа BinaryTree. А собственно дерево, тот корень, который корень-корень, только инициирует рекурсивный обход.

Вот как-то так. Ещё раз подчеркну, главное – чтобы рекурсия прошла легко, без напряга, естественно; рекурсия – это хорошо, легко и просто – таков лейтмотив этого занятия. Чтоб она детям понравилась.

О чём мы не сказали

А не говорили мы совсем об удалении вершины. Это вполне себе исполнимо, но там необходимо гораздо более глубокое понимание рекурсии, чем есть сейчас. Поговорим об этом обязательно, но попозже, когда мы таки погрузимся в рекурсию значительно сильнее, а мы погрузимся.

А ещё мы не говорили об эффективности этой структуры, хотя и понятно, чисто на пальцах, что такая сортировка, а у нас получилась таки сортировка, работает побыстрее сортировки вставками со списком. Для совсем уж эффективности надо бы ещё и балансировать дерево, делать его “поуравновешенне”, не допускать длинных путей вниз. И это целая отдельная и непростая тема. Поговорим о ней попозже. Также, как и о других эффективных алгоритмах сортировки.

Практика.

И соответственно давать задания на практику. Опять всё про бинарные деревья, и притом легко формулируемые в терминах рекурсии.

1. Написать метод, вычисляющий высоту дерева.
   Высота дерева – это количество вершин на пути от корня до наиболее удалённой вершины. Пишем рекурсивную функцию: высота пустого дерева = 0; высота непустого = максимум из высот двух поддеревьев + 1.
2. Написать методы, которые вычисляют
   a. количество вершин в дереве
   b. количество листьев (вершин без потомков) в дереве
   c. количество вершин, у которых ровно один сын
   d. количество вершин, у которых ровно два сына
3. Написать метод

```
func (s BinaryTree) CountX (x int) int
```

который вычисляет, сколько раз ключ x содержится в дереве. тут тоже надо использовать особенность именно дерева поиска, и не обходить всё дерево.

4. Написать функцию, определяющую количество вершин дерева на слое номер K (т.е. вершин, отстоящих от корня на K рёбер). Опять-таки: в пустом дереве таких вершин 0, в непустом – количество вершин на слое номер (K-1) в левом поддереве + количество вершин на слое номер (K-1) в правом поддереве.
5. Метод, создающий и возвращающий копию дерева.

```
func (s BinaryTree) Copy () BynaryTree
```

Именно второй экземпляр дерева, занимающий свою память. Не просто вернуть указатель на тот же корень.

6. Написать метод

```
func (s BinaryTree) Distance (x int) int
```

который возвращает расстояние от корня до ближайшего узла, содержащего ключ x. Если такого ключа в дереве нет, то возвращает -1.

7. a. Написать метод,

```
func (s BinaryTree) Sup (x int) int
```

который возвращает наименьшее число, большее x.
b. Написать метод,

```
func (s BinaryTree) Inf (x int) int
```

который возвращает наибольшее число, меньшее x.

8. Хорошая задача на подумать. Мы видим, что вид дерева поиска зависит от того, в каком порядке в него вставляются числа. Например, если числа идут в порядке возрастания, то у нас получится, что все ветки идут от корня вправо - полученная конструкция скорее похожа на список. А задание такое. Сбросить ключи дерева в файл в таком порядке, чтобы при считывании чисел из файла и вставке их в дерево, получалось первоначальное дерево. Причём само собой получалось, только из-за порядка следования элементов, никакой другой информации в выходном файле хранить не разрешается.

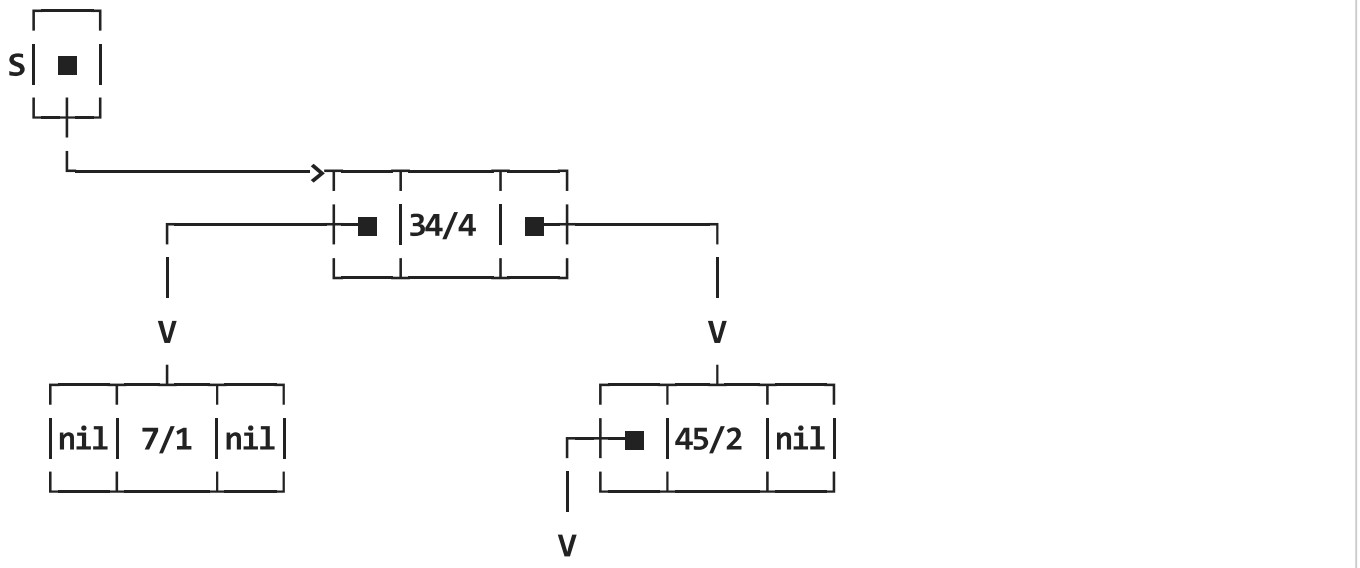
> Спойлер. Тут спасает например вывод ключей в инфиксном порядке: сначала выводим ключ из корня дерева, а потом точно также, рекурсивно выводим левое поддерево, а потом правое. Понятно, что этот порядок неединственный, но он работает, а код получается очень простой.

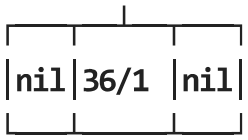
Такие задания-упражнения можно придумывать ещё долго и много, но пока хватит, хотя добавления привествуются.

А напоследок ещё одна отдельная задача, собственно, это задача но построение несколько иной структуры данных, хотя и минимально отличающейся от того, что у нас уже есть.

Бинарное дерево поиска со счётчиками.

Рассмотрим такую задачу. Пусть K - заданное число, не превосходящее количества элементов в дереве. Требуется найти найти K-е по порядку число. Т.е.то число, которое стояло бы на K-ом месте, если бы мы отсортировали все ключи дерева.
Вроде бы сортировка есть, и кажется, что K-е число можно найти как-то побыстрее, чем просто перебирать все ключи в порядке возрастания. Но оказывается, что всё равно приходится перебирать все числа, начиная с наименьшего (просто, чтобы их подсчитать). А давайте мы немного изменим структуру - в каждый узел добавим размер поддерева, корнем которого является это узел, т.е. количество вершин в дереве (поддереве), корнем которого является этот узел. Вот таким образом - в каждом узле хранятся значение ключа и размер поддерева:





Задания.

1. Изменить метод вставки нового элемента соответствующим образом. Изменения получаются минимальные.
2. Написать методы, которые подсчитывают количество узлов в дереве, значение ключа в которых
   a. меньше X
   b. больше X.
3. Ну, и, конечно, написать метод, который возвращает K-й по порядку ключ в дереве (интеллигентно это называется K-я порядковая статистика).
   При этом мы запросто можем не обходить всё дерево. Идея такая. Вот мы стоим в корне дереве (поддерева) и ищем K-й по порядку ключ. Пусть P - размер левого поддерева нашего дерева. Тогда
   - если P = K-1, то возвращаем ключ в корне
   - если P > K-1 (P >= K), то ищем K-й по порядку ключ в левом поддереве
   - если P < K-1, то ищем (K - (P+1))-й по порядку ключ в правом поддереве. Просто сказка.